Early Load Address Resolution Via Register Tracking

Michael Bekerman[1], Adi Yoaz[2], Freddy Gabbay[3], Stephan Jourdan[2], Maxim Kalaev[2] and Ronny Ronen[2]

Intel Corporation

Abstract

Higher microprocessor frequencies accentuate the performance cost of memory accesses. This is especially noticeable in the Intel's IA32 architecture where lack of registers results in increased number of memory accesses. This paper presents novel, non-speculative technique that partially hides the increasing load-to-use latency, by allowing the early issue of load instructions. Early load address resolution relies on register tracking to safely compute the addresses of memory references in the front-end part of the processor pipeline. Register tracking enables decode-time computation of register values by tracking simple operations of the form reg±immediate. Register tracking may be performed in any pipeline stage following instruction decode and prior to execution.

Several tracking schemes are proposed in this paper:

- *Stack pointer tracking allows safe early resolution of stack references by keeping track of the value of the ESP register (the stack pointer). About 25% of all loads are stack loads and 95% of these loads may be resolved in the front-end.*
- *Absolute address tracking allows the early resolution of constant-address loads.*
- *Displacement-based tracking tackles all loads with addresses of the form reg±immediate by tracking the values of all general-purpose registers. This class corresponds to 82% of all loads, and about 65% of these loads can be safely resolved in the front-end pipeline.*

The paper describes the tracking schemes, analyzes their performance potential in a deeply pipelined processor and discusses the integration of tracking with memory disambiguation.

1. Introduction

One of the key factors influencing microprocessor performance is the load-to-use latency, i.e. the time required for a load instruction to fetch data from the memory hierarchy and to deliver it to the dependent instructions. With higher microprocessor frequencies, the relative performance cost of memory accesses is increased. First-level cache accesses are expected to consume two to five cycles in next-generation processors.

This paper introduces a new technique to decrease the load-to-use latency. This technique is similar to load-address prediction in the sense that it determines the address of a load instruction during an early stage of the pipeline in order to initiate the memory access earlier. However, this early load address resolution scheme is safe and does not rely on the history of previous accesses. The proposed scheme is based on the fact that most addresses can be safely computed at decode time by using a new register-value tracking mechanism. Register tracking enables early computation of the values of general-purpose registers by tracking simple operations of the form reg±immediate. We present three flavors of early load address resolution: absolute load tracking, stack accesses tracking, and tracking of all loads with addresses of the form reg±immediate. We also address the issues of load-store disambiguation and multiple address resolutions per cycle, required for implementation of early address resolution in a pipelined super-scalar processor.

The remainder of this paper is organized as follows. Section 2 presents the register tracking and the early load address resolution schemes. It also discusses the impact of early load-address resolution on memory dependencies. Section 3 describes ways to accommodate multiple accesses to the memory hierarchy. Section 4 provides both statistical and performance results. Related work is discussed in section 5 and concluding remarks are given in section 6.

2. Register Tracking

2.1 Motivation

The IA32 instruction set architecture uses a *procedure stack* to pass parameters to functions and to hold automatic (local) variables. About 25% of all memory load operations access the procedure stack. Stack loads differ from other load operations in their address components: the ESP register and a displacement specified by the instruction opcode[4]. The addresses of stack loads depend only on the value of the ESP register. Modifications to the ESP register usually have a simple form, $ESP \leftarrow ESP \pm immediate$, and thus can easily be tracked at decode time. As a result, addresses of most stack references may safely be calculated after the instruction is decoded. Early address resolution allows the issuing of stack loads earlier, improving the load-to-use latency and increasing the ILP.

We focus on the ESP-based stack accesses in Sections 2.2 and 2.3, and extend the concept to include absolute addresses and all general-purpose registers in the later sections.

[1] Currently at HAL Computer Systems, bekerman@hal.com

[2] {adi.yoaz, stephan.jourdan, maxim.kalaev, ronny.ronen}@intel.com

[3] Currently at Mellanox Technologies Inc, fredg@mellanox.co.il

[4] EBP-based stack addressing is discussed in Section 2.5.

2.2 Decode-time computation of ESP value

The ESP register in the IA32 instruction set architecture holds the procedure stack pointer and is almost never used for any other purpose. The ESP register can be modified implicitly using instructions such as CALL/RET, PUSH/POP and ENTER/LEAVE, which add or subtract an immediate value from ESP [12]. The ESP register can also be modified explicitly, as a general-purpose register, almost always in the form of *ESP ← ESP ± immediate*.

Rarely, inter-privilege level calls and calls to interrupt or exception handlers switch the procedures stack. These instructions either explicitly load a new value into ESP or perform a more complex arithmetic operation on ESP.

Regular procedure stack references in the IA32 architecture are performed through the PUSH/POP/MOV instructions. These instructions are translated by the instruction decoder into RISC-like *micro-operations* (uops) as shown in figure 2.1. The value of the immediate operand is explicitly provided by the uop opcode. Our simulations show that about 95% of all ESP modifications are of the form *ESP ← ESP ± immediate*, as will be shown in Section 4.

PUSH EAX	ESP ← ESP - imm mem[ESP] ← EAX
POP EAX	EAX ←mem[ESP] ESP ← ESP + imm
Load EAX from the stack	EAX ← mem[ESP+imm]

Figure 2.1: Micro-operation sequences of regular IA32 stack operations

These ESP modifications can easily be tracked in any pipeline stage after instruction decode [18]. Furthermore, once the ESP value is known at some point, the exact value of the ESP register may be computed after each consecutive instruction is decoded.

We propose a new hardware mechanism to track ESP register values and ESP-based stack memory references. We place this ESP register tracking hardware in the pipeline stage right after the instruction decode. The tracker inspects each uop to detect uops modifying the value of ESP as well as memory operations using ESP for address calculation. Note that the ESP tracker processes uops in program order.

The ESP tracking hardware uses a special register to record the current value of the ESP stack pointer, referenced as CESP. Whenever a simple uop modifying the ESP value is detected, this value is used to safely calculate the new ESP value without waiting for this uop to reach the execution stage of the pipeline. The tracker copy of the last ESP value (the CESP) is then updated. All subsequent stack operations may use this new calculated value of ESP. If all modifications are trackable, CESP reflects the correct ESP value updated for all uops that have passed the tracking stage of the pipeline.

The conditions required for a simple ESP modification to be trackable:

1. ESP is the destination register.
2. The uop type is ADD, SUB or MOV-immediate.
3. ESP is one of the sources of the operation (null for MOV-immediate).
4. The second source operand is an immediate value.

Figure 2.2 shows the structure of the ESP tracking mechanism. First, it checks that the uop opcode is one of the trackable opcodes, and that the source operands are ESP and immediate values. Then it selects the immediate source operand and uses the uop opcode to select the correct operation on the CESP and the immediate value. Once the ESP modification is successfully tracked, the new calculated ESP value is written into the CESP register.

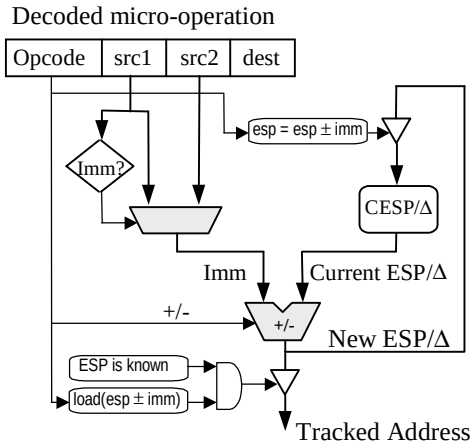

Figure 2.2: ESP tracking and ESP-based address calculation mechanism

When the ESP tracker detects an ESP-modifying uop that does not satisfy one of the conditions, the new ESP value cannot be calculated. Such uops include complex operations on ESP, loads from memory into ESP, and operations involving other registers. In these cases, the stack tracking hardware sets the ESP_unknown flag and records the sequence number identifier of the untrackable operation. A new status bit is attached to each micro-operation that indicates whether the current operation modifies the ESP value in an untrackable way. For every executed uop, the execution core checks this status bit and forwards the result to the tracker if the bit is set.

When the stack tracking hardware detects an untrackable ESP change, it waits until this uop passes the execution stage of the pipeline and the new ESP value is received. During that time interval additional uops modifying the ESP value could have passed the tracking stage. The ESP value received from the execution core does not reflect these changes. Therefore, when the ESP_unknown flag is set, the stack tracker must continue tracking the ESP changes calculating the ESP change delta (Δ), as shown in figure 2.2. The ESP delta provides the cumulative change to the value of ESP by all uops that passed the tracker since the ESP_unknown flag had been set. CESP register is reset by setting the ESP_unknown flag and is then used to keep the ESP delta. The tracking operation is otherwise identical to the ESP tracking. The delta value is added to the ESP value received from the execution core to obtain the new up-to-date ESP value and is then stored to CESP.

If the ESP value was successfully tracked, there is no need to schedule and execute again the tracked uop. Indeed, the correct instruction result was already produced by the tracker. Hence,

ESP tracking effectively increases the execution bandwidth and can save physical registers, reservation station entries, etc. Implementation-dependent modifications required in this case to register renaming and recovery mechanisms are beyond the scope of this paper.

When a branch misprediction is detected, the correct CESP value must be recovered. One option is to copy the value of the recovered architectural ESP register itself. If a physical register holding the last copy of ESP is ready, the register file is read and the value is forwarded to the tracker. Otherwise, the ESP_unknown flag is set and the tracker starts recording the new delta. Another option is to checkpoint the CESP/Δ value and the ESP_unknown bit on every branch. If the ESP was known before the branch misprediction (the checkpointed ESP_unknown flag wasn t set), the CESP value is recovered immediately. Otherwise, the tracking restarts after the ESP-modifying operation from the correct path executes and forwards its result to the tracker.

Note that more than a single instruction modifying the ESP value may be decoded in a single cycle. The tracker must handle a number of concurrent ESP modifications per cycle. The stack tracking hardware employs a two-level structure similar to register renaming of a number of dependent instructions. For each incoming ESP modification, the first stage selects the correct initial CESP value. This may be either the CESP value from the previous cycle or an outcome of one of the earlier CESP calculations performed at the same cycle. The second stage performs the actual CESP calculation using the correct initial CESP. Timing constraints may require a pipelined tracker implementation with the corresponding bypass paths. However, processor cycle time should not be affected.

2.3 Early stack loads resolution

Knowledge of the current ESP value may be utilized to calculate the effective addresses of stack loads after the instruction decode stage in the pipeline. Early resolution of stack load addresses allows issuing stack loads ahead of their dependent instructions, reducing or eliminating the load-to-use latencies of accessing the L1 or the L2 data cache. Early stack load resolution improves instruction parallelism, shortens critical paths, and boosts performance, similarly to load address prediction, as described in [11,4].

The advantage of the early load address resolution through ESP tracking over the address prediction schemes is that it is safe[5]: no verification or recovery schemes are required. Indeed, the addresses of stack loads are calculated ahead of time, based on the decode-time knowledge of the address components and the ability to track the values of these components. No speculation is applied. Early stack load resolution is significantly simpler and more cost effective compared to other techniques for reducing the load-to-use latency. On the other hand, ESP tracking handles only a subset of all loads, and these loads are likely to hit in the L1 cache. Therefore, the ESP tracking potential gain is lower than that of aggressive address prediction schemes. Furthermore, memory renaming schemes, such as in [16,13], are likely to address a significant portion of the loads processed by ESP tracking.

Load effective addresses in the IA32 architecture are calculated as Base + (Index * Scale) + Displacement. Any of these components can be omitted. Both the Base and the Index components are specified by general-purpose registers, while ESP may serve only as the Base. Typical stack loads specify the simplest addressing mode: ESP register is the Base while the Index component is omitted. In this case, the load effective address is simply the sum of the ESP value and the Displacement specified in the instruction opcode. Tracking only these simple cases captures nearly all stack references.

When the tracker recognizes an ESP-based load, it uses the CESP value and the displacement specified in the load opcode to calculate the load effective address. Load address calculation involves exactly the same operations as the tracking of an ADD-immediate uop. As such, we use the same hardware for the ESP register tracking and the early load address computation. For load uops, the outcome of the addition of CESP and the immediate displacement values provides the load effective address and it is not written into the CESP register, as shown in figure 2.2.

The calculated address is used to initiate a data cache access in the front-end stages of the pipeline. The data cache access of the resolved load overlaps with the original load instruction progress in the pipeline up to the Reorder Buffer allocation stage. Since early load address resolution is safe, the original load does not consume additional scheduling or memory bandwidth. Instructions dependent on the early resolved load are executed based on the early loaded data as soon as the cache access completes.

Stack load resolution mechanism may use a dedicated port to the data cache, in this case the tracked loads may proceed in parallel to other loads. To reduce the implementation cost, one may opt to share the data cache ports and arbitrate between different requests from both the tracker and the regular memory pipeline. A separate stack cache allows stack load accesses to be performed in parallel to other loads while eliminating the cost of additional ports to the data cache. Stack cache implementation is discussed in Section 3.

When the tracking hardware is unable to track the ESP value, no early load address calculation is performed, and all stack references execute as usual.

The stack tracking mechanism employs a state machine consisting of two states:

- **ESP value is known**. Addresses of stack loads are resolved in the front-end.
- **ESP value is unknown**. Stack addresses cannot be resolved in the front-end.

There are several events that cause the state machine to change its state and perform certain actions. Four events are triggered by incoming decoded uops:

1. Loads with an address component consisting of an ESP and an immediate operand.
2. Loads with an address component other than (1).
3. Simple trackable ESP-to-immediate arithmetic operations that modify the ESP.
4. Operations other than (3) that modify the ESP.

[5] The issue of load-store disambiguation is addressed in Section 2.6.

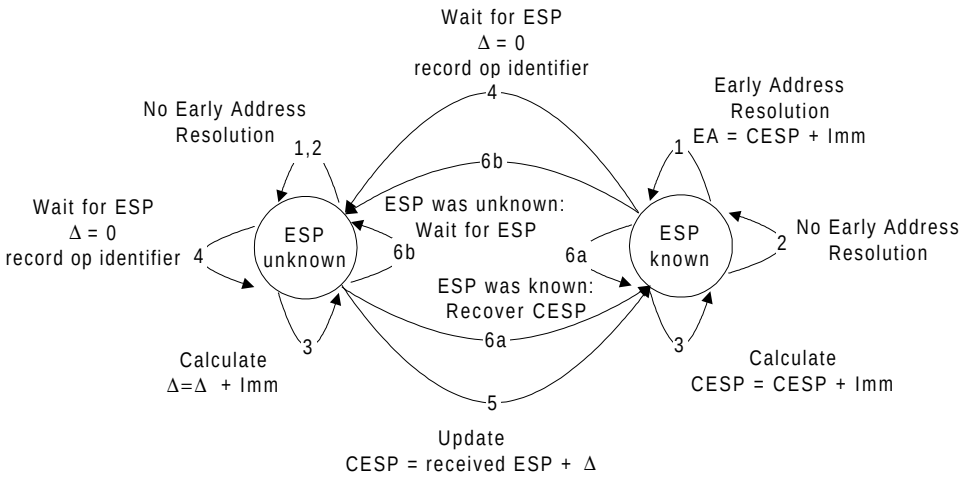

Figure 2.3: Stack Address Tracking State Machine

There are several asynchronous events triggered by the execution core:

5. A new ESP value is received from the execution core after an untrackable ESP modification is executed.

6.a A pipeline flush after a branch misprediction when ESP was known at the branch, or

6.b A pipeline flush after a branch misprediction when ESP was unknown at the branch

Figure 2.3 shows the ESP tracking state diagram with the ESP known and ESP unknown states and arc numbers corresponding to one of the six events above. Whenever the current ESP value is known and an ESP-based load is recognized (**event 1**), the load effective address is calculated and the load is issued to the cache. No early address resolution is performed for other loads (**event 2**). For simple trackable operations, the new CESP value is calculated based on the tracked CESP value and the instruction s immediate value (**event 3**). When an untrackable ESP modification (**event 4**) is recognized, the ESP tracker switches to the ESP unknown state. No early address calculation is performed in this state and the ESP delta is computed. If an additional untrackable uop is encountered (**event 4** in the ESP unknown state), the new uop identifier is recorded and the delta value is reset. When an untrackable uop that matches the recorded identifier is executed (**event 5** in the ESP unknown state), the calculated ESP value delivered by the execution core, is added to the ESP delta to form the correct new CESP value. In the case of a pipeline flush, if the value of CESP was known before the branch (**event 6a)**, it is recovered from the checkpoint. This is shown by two arcs labeled 6a leading from both states to the ESP known state of the state machine. If the ESP was not known before the branch (**event 6b)**, the tracker moves to the ESP unknown state, as shown by two arcs labeled 6b leading from both states to the ESP unknown state.

2.4 Absolute loads

The ESP tracking mechanism may be used to handle loads with absolute addressing mode. The addresses of such loads are composed of a single component: the displacement. These loads constitute about 10% of all loads in the IA32 architecture. The address of these loads may be resolved in the front-end and the loads may be issued earlier using the existing address tracking mechanism. The only change to the tracker structure is to allow a null Base component of the address in addition to ESP. The gain from early accessing of absolute loads is smaller since these loads do not depend on any previous instruction.

2.5 Early resolution of all loads

The same mechanism used for tracking of the ESP register may be used for tracking values of all architectural registers. In this case the tracker must use a Latest Register Values (LRV) table to record the tracking-time values of all architectural registers. Whenever a simple operation on any of the known registers is recognized, the tracker performs the calculation, updates the LRV, and marks the destination register as known.

Since more than a single register is tracked, the tracker must handle uops with the destination register different from the source register(s). The extended list of trackable uops in this case is *Reg2* ← *Reg1* ± *immediate*, where *Reg1* and *Reg2* may be any register[6].

The main disadvantage of this scheme is its underlying hardware complexity. The number of computations that have to be performed every cycle is equal to the decoder bandwidth[7].

[6] Frequent uops of this form in the IA32 microarchitecture are:

1. Reg2 ← ADD/SUB (Reg1, Imm)
2. Reg2 ← LEA (Reg1, Imm)
3. Reg2 ← MOV (Reg1)
4. Reg2 ← MOV (imm)
5. Reg2 ← XOR (Reg1, Reg1), that nullifies Reg2.
6. Reg2 ← OR (Reg1, Reg1) effectively moves Reg1 into Reg2

The last two private cases are extensively used by IA32 compilers. Handling these cases significantly increases the overall register trackability.

[7] While ESP register tracking theoretically requires the same number of computations per cycle, a number of simplifying assumptions may be made without hurting the performance, for example, no more than 3 ESP modifications per cycle with 6-wide decode, or resolving only part of the loads.

Furthermore, untrackable ESP modifications are rare, and are usually distant, in the program order, from most stack loads, giving the tracker a chance to catch up. On the other hand, the dynamic distance between the instructions modifying a general-purpose register and consecutive loads using this register for the address computation is usually small, reducing the rate of early resolutions as the pipeline becomes longer.

Note that if a register is unknown, no load addresses based on that register are resolved by the tracker, resulting in lost opportunities, but not in mispredictions.

Finally, IA32 16-bit code extensively uses the BP register (16-bit version of EBP) to point to stack elements. To reduce the implementation cost, we may choose to track only the ESP and the EBP registers. On Windows™ applications this scheme handles about 8% more loads than ESP-only tracking.

2.6. Memory Dependencies

Early load address resolution issues loads to the cache in the front-end part of the pipeline, ahead of other memory operations. However, a load cannot be advanced ahead of a preceding, in the program order, store to the same address. Hence early load address resolution should be coupled with a dynamic memory disambiguation mechanism. One possibility is to use existing memory disambiguation scheme [19] or dependency prediction mechanisms [16,7]. However, this requires additional address ports and control paths and increases the number of disambiguation operations that must be performed per cycle.

We propose to take advantage of the existing tracking mechanism to resolve part of the memory dependencies incurred by early load resolution. We use the tracking mechanism to early resolve store addresses the same way it resolves load addresses. Generation of load and store addresses in the front-end part of the pipeline provides the information necessary to determine whether an early resolved load can safely bypass prior stores. In the case of untrackable store addresses, we adopt the conservative approach and do not allow succeeding trackable loads to bypass such stores.

3. Stack Cache

Successful load address tracking triggers an early data cache access. However, front-end initiated accesses compete with other memory references for the data cache ports. In the context of our ESP tracking scheme, separating the stack accesses from all other load accesses is important. This introduces two major advantages: first, several smaller caches are faster than a single large cache and second, each of these distributed caches may work in parallel without requiring additional ports. Decoupling the stack references from all other memory references is straightforward: only loads/stores, which use ESP as their source address operand, are redirected to a separate small *stack cache* [8,6].

The stack cache and the L1 data cache maintain a mutually exclusive data. This is achieved by maintaining a snooping mechanism to detect incorrect data placement. When an incorrect data allocation is found, the data is transferred to the appropriate cache. Experimental measurements show that our stack cache redirection criteria is very accurate only 0.5% (on average) of the stack accesses are wrongly directed to the (64-KB) L1 cache, while 0.7% of non-stack references are incorrectly sent to the (4-KB) stack cache. In order to maintain memory consistency, the stack cache as well as the L1 cache accesses are snooped. Since the number of wrong redirections between the caches is very low, even a costly penalty can be tolerated in case of wrong redirections.

Performance simulations indicate that a small 4-Kbyte stack cache achieves a hit rate of over 97%. This result shows that the stack cache can efficiently take advantage of the memory locality of stack references. Since the stack cache is relatively small, it may be designed for lower latency, reducing load-to-use delays even in case of untrackable loads.

4. Simulation Results

This section analyses the performance of register tracking and early load address resolution scheme and evaluates its impact on overall processor performance. Section 4.1 presents our simulation methodology. Section 4.2 identifies the loads that may potentially benefit from our tracking scheme and classify them into several subsets. In Section 4.3 we analyze each set of loads and present performance measurements.

4.1 Simulation Methodology

Results provided in this paper were collected from an IA32 trace-driven simulator. Each trace consists of 30 million consecutive IA32 instructions. Traces have been selected to be representative of the entire execution, and they record both user and kernel activities. Results are reported for 45 traces grouped into 8 suites as summarized in table 4.1

Benchmark suite	Notation	Number of traces
SPECint95	INT	8
CAD programs	CAD	2
Multimedia applications using MMX instructions	MMX	8
Games, e.g. Quake	Games	4
Java Programs	JAVA	5
TPC benchmarks	TPC	3
NT common programs, e.g. Word	NT4	8
Windows 95	W95	7

Table 4.1 Benchmark suites.

In this study we used a detailed cycle-by-cycle performance simulator modeling a processor featuring:

- 6-wide out-of-order execution engine.
- 8-stage pipeline from decode to execution.
- 128-entry instruction window.
- 4 integer, 1 complex, 2 load/store and 1 FP units.
- Trace Cache based front-end with the configuration presented in [17].
- A 16-bit history GSHARE branch predictor.
- 64KB on-die L1 data cache, 4-cycle hit-latency.
- 1MB L2 cache size, 14-cycle hit-latency.
- 4-KB, 4-way set-associative Stack Cache, 4-byte line size, 1 cycle hit-latency.
- Instruction latencies are common to Intel s processors.

Register tracking (either ESP or general purpose registers) as well as early load address resolution are performed right after the decode stage. We have simulated a deeply pipelined machine with a latency of 8 clock cycles from the first decode stage till the first execution stage, while the first level cache latency is 4 cycles.

4.2 Classifying potential loads

Our analysis examines three different sets of load memory references. The first set consists of stack references, i.e., loads using ESP register as a Base component of effective address. We term these loads *ESP-based loads*. The second set consists of loads using an absolute addressing mode, termed *absolute loads*. The third set consists of loads with a displacement-addressing mode (reg ± immediate). In this case the Base register can be any general-purpose registers including the ESP. We term these loads *displacement-based loads*. Table 4.2 summarizes the distribution of these sets out of the total number of loads. We observe that the dynamic distribution of ESP-based loads varies from 18% in MMX to 39% in Java. The reason for the extensive use of stack segment by Java traces is the implementation of the JAVA virtual machine as a stack machine.

The average distribution of absolute loads is quite significant it varies from 5% in Java to 16% in Games, yielding an average distribution of 10%. The last set of loads represents the most common addressing mode with an average distribution of more than 80% (including the ESP-based loads).

Benchmarks	Load reference type		
	ESP-based	Absolute	Disp.-based
CAD	38%	2%	88%
Games	26%	16%	77%
INT	28%	15%	77%
JAVA	39%	5%	89%
MMX	18%	13%	70%
NT4	24%	6%	88%
TPC	22%	9%	85%
W95	23%	7%	87%
Average	25%	10%	82%

Table 4.2 The distribution of the three sets of loads.

4.3 Early address resolution of ESP-based loads

In order to evaluate the potential of early resolution of stack references, we first examine what portion of ESP modifications can be tracked. An ESP-modifying instruction is trackable if it has the form $ESP \leftarrow ESP \pm immediate$ and the previous ESP value was known. Figure 4.1 presents the percentages of trackable ESP modifications out of the total number of instructions modifying the ESP. We observed that in all of our benchmarks, ESP modifications are highly trackable. Trackability percentages vary from 87% in Windows95 to nearly 100% in SPECint95. The relatively low trackability in Windows95 stems from the large number of privilege level switches (user and kernel modes) that change the frame of the stack segment and modify the ESP value in an untrackable way.

Note that all the results in Section 4 take into account pipeline effects, that is, all instructions passing the tracking stage when the ESP value is unknown are considered non-trackable.

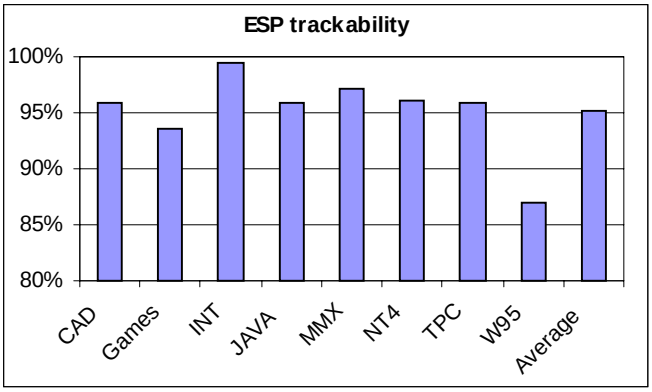

Figure 4.1 The percentages of trackable ESP modifications.

As table 4.2 indicates, on average 25% of all loads are ESP-based loads. The target of the ESP tracking scheme is to perform early address resolution of these loads. However, not all ESP-based loads can be early resolved, since no address calculation is performed when the ESP value is unknown. Early adddress resolution resumes only after the new ESP value is received from the core. Figure 4.2 shows that an average of 95% of ESP-based loads can be resolved in earlier pipeline stages. Note that the percentage of early resolved ESP-based loads in figure 4.2 is proportional to trackability of ESP modifications, shown by figure 4.1. Due to this correlation, Windows95 traces exhibit relatively small percentages of ESP-based loads whose addresses can be early resolved, while almost 100% of stack loads of SPECint95 traces are resolved

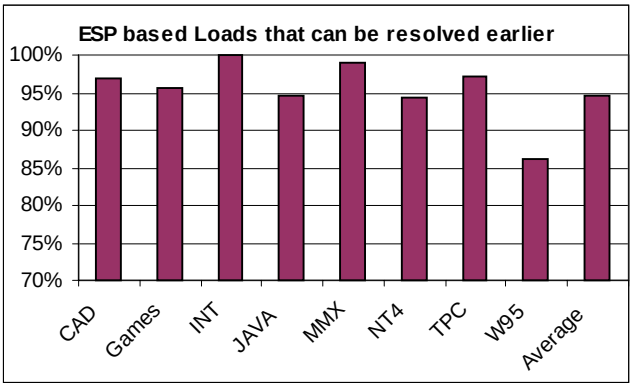

Figure 4.2 ESP-based load whose addresses may be early resolved.

4.4 Early address resolution of displacement-based loads

This subsection examines the generalized tracking scheme, which monitors modifications to all general-purpose registers as

well as the ESP. Figure 4.3 illustrates the percentage of trackable modifications to the IA32 general-purpose registers out of all register modifications. As much as 42% (on average) of all general-purpose register modifications can be tracked after instruction decode. The figure indicates that general-purpose register trackability, though lower than ESP, is significant and can be exploited for early address resolution.

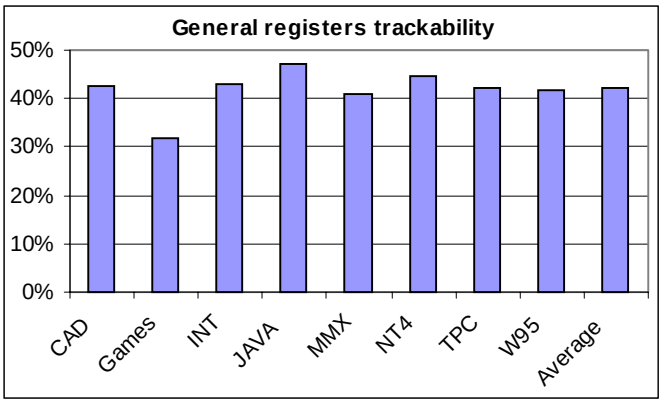

Figure 4.3 The percentages of trackable modifications to general-purpose registers.

Figure 4.4 illustrates the portion of all displacement-based loads that can be resolved in the front-end part of the pipeline using general-purpose register tracking. We found that on average 66% of the loads that use displacement addressing mode are trackable.

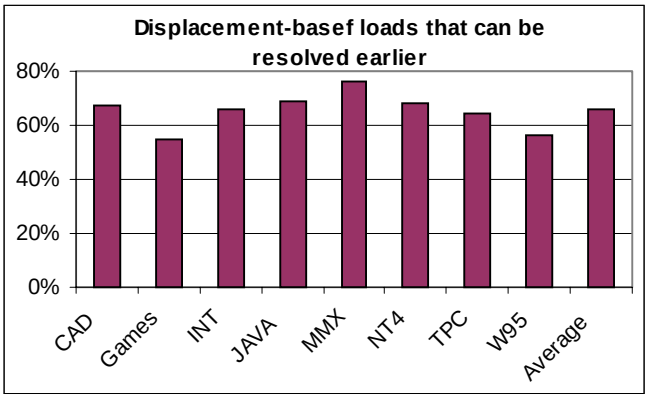

Figure 4.4 Displacement-based loads, whose addresses may be early resolved.

4.5 Limitations due to memory ambiguity

The gain of the early address resolution scheme comes from advancing load accesses to memory. However, a load whose address was early resolved cannot be advanced ahead of a preceding store that accesses the same address. Figure 4.5 shows the amount of early resolved loads that collide with preceding stores. The amount of collisions is presented for the three sets of early-resolved loads: ESP-based, displacement-based and absolute-based loads. The average collision rate is 46%, 39% and 11% respectively. The high fraction of collisions in the stack references arises from the nature of the stack operations exhibiting short-term producer-consumer dependencies.

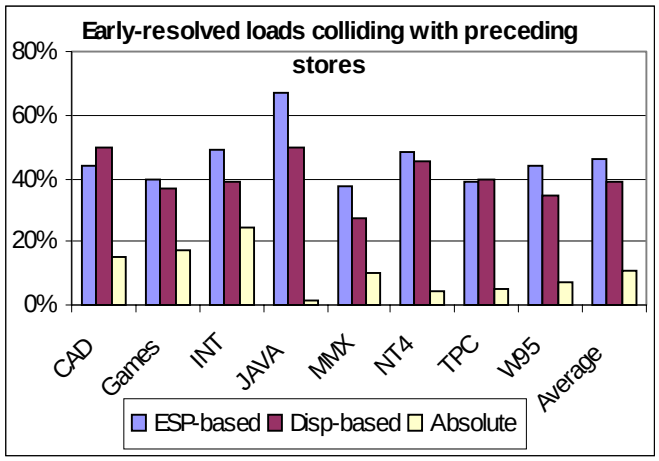

Figure 4.5 Percentage of early-resolved loads colliding with preceding stores.

In order to advance loads as much as possible without violating memory-ordering constraints, we consider using two kinds of disambiguation mechanisms. The first tracks the values of store address operands the same way we do for loads. As a result, reference addresses of stores can be resolved earlier. By using this information we can decide which loads are eventually advanced. We classify store instructions into three sets, the same way as we did for the loads in section 4.2: ESP-based stores, displacement-based stores and absolute stores. We observed that the average dynamic distribution of each set of stores (out of the store instruction count) is 46%, 87% and 6% respectively.

Figure 4.6 presents the portion of ESP-based and displacement-based stores that can be early resolved by our tracking scheme.

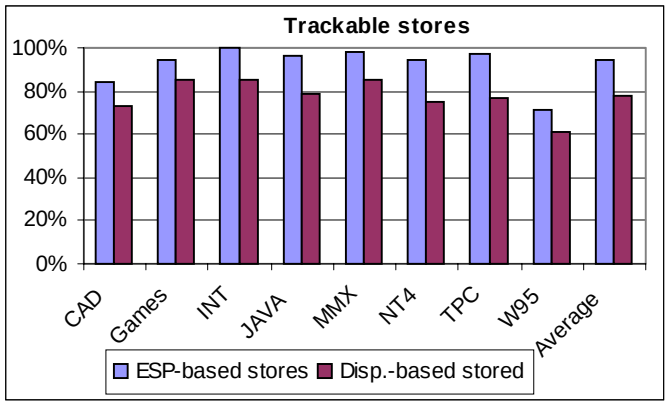

Figure 4.6 Percentages of trackable stores.

Our measurements show that about 95% of the ESP-based stores are trackable. We opt to lose the opportunity of advancing loads beyond untrackable ESP-based stores (5%) in order to save the collision penalty. Our simulation shows that the collision rate of the ESP-based loads with the non ESP-based stores is as low as

0.6% on average (not shown on the graph). Hence, simply tracking ESP stores in addition to ESP loads allows advancing the non-colliding tracked loads with only a 0.6% error rate, without requiring a complex disambiguation mechanism. Figure 4.6 also shows that the displacement-based stores exhibit lower trackability ratio of 78%. Therefore, for this kind of stores we prefer to use the memory disambiguation mechanism proposed in [19].

4.6 Performance evaluation

This section evaluates the impact of early address resolution scheme on processor performance and analyses its causes.

First, we examine the number of regular integer ALU instructions that can be resolved by register tracking. Figure 4.7 presents the percentage of ALU instructions (out of the total instruction stream) that can be safely executed in the front-end by the register tracking mechanism. In the case of ESP-based tracking, 7% (on average) of the entire instruction count can be eliminated from the normal execution pipeline (saving execution bandwidth). When tracking the values of all general-purpose registers, as many as 21% of the total instruction count is early executed.

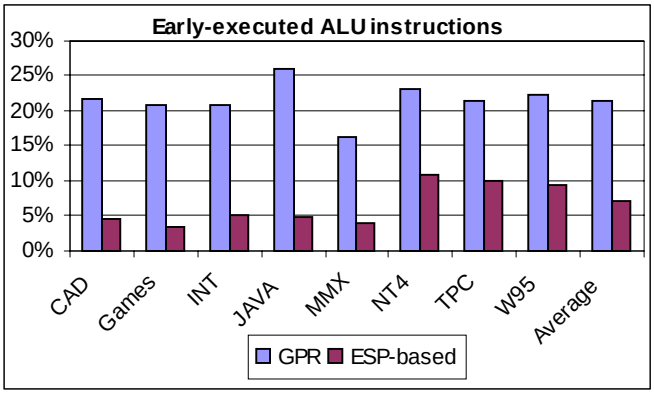

Figure 4.7 Early-executed ALU instructions.

Second, we examine the average *load hoisting interval* the time interval in which a load memory access was advanced by the tracking scheme[8]. The load hoisting distance is mainly affected by the distance between the load and the potentially colliding preceding stores. Figure 4.8 presents histograms of the average hoisting interval for all early-resolved loads. We observe that in most applications from 50% to 70% of early resolved loads are advanced more than 15 cycles, masking the first and the second-level caches accesses.

[8] We approximate the *load hoisting interval* as the number of cycles between the beginning of the cache access for the advanced load and the time when all load dependencies are satisfied. Note that this takes into account register dependencies as well as memory dependencies. The approximation comes from the fact that we disregard resource conflict delays, therefore these are lower-bound numbers.

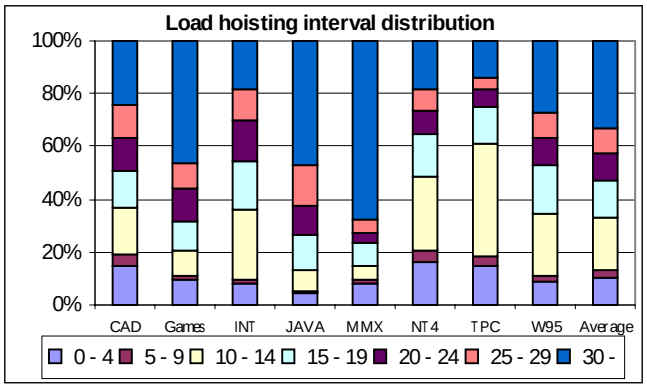

Figure 4.8 Distribution of load hoisting interval.

The third parameter affecting the impact of early resolution on processor performance is the fraction of advanced loads that result in an early utilization of their results. These loads are termed *useful early-resolved loads*. Notice that not all of the early resolved loads are useful since the dynamic instruction distance [10] to their data-consuming instructions may be too long. Figure 4.9 illustrates the percentages of useful early-resolved loads out of all early-resolved loads. It shows that the average percentages for the ESP-based, displacement-based and the absolute loads are 46%, 41% and 60% respectively. In the case of the absolute loads, the dynamic instruction distance is relatively short, therefore these early-resolved loads are better exploited. The percentages of useful ESP-based loads are higher than displacement-based loads mainly because of the nature of stack operations. Such operations tend to exhibit tight dynamic instruction distances relative to other displacement-based loads.

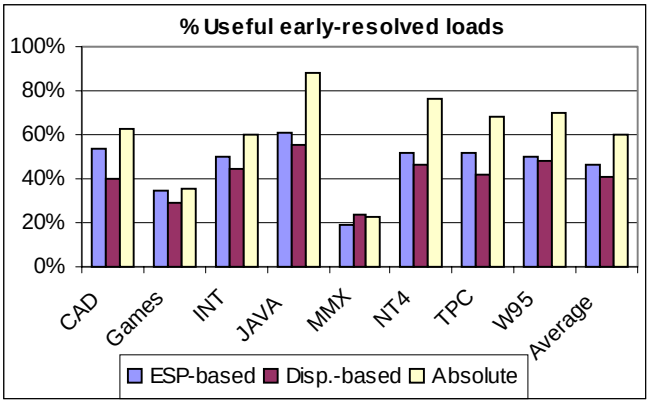

Figure 4.9 Percentages of useful early-resolved loads.

Finally, figure 4.10 presents the overall performance speedup achieved by the early address resolution scheme. The average performance improvement of ESP load and store tracking is about 7%. For the displacement-based load tracking the average performance speedup reaches 12%, while the absolute load tracking gain less than 1%. We observe that the MMX benchmarks deliver the lowest performance improvement both for the ESP-based and the displacement-based loads due to the fact

that they exhibit a relatively small fraction of useful early-resolved loads.

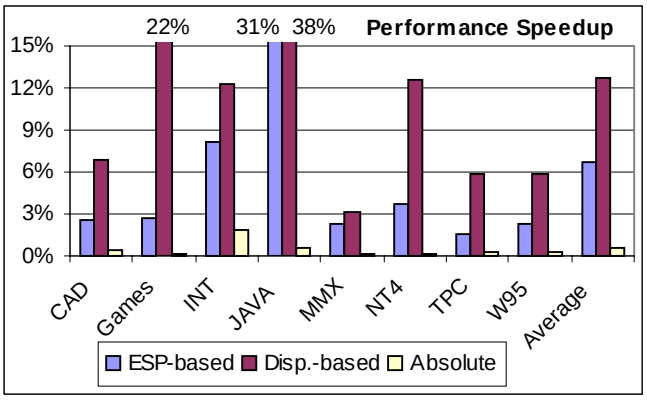

Figure 4.10 Performance improvement.

5. Related Work

Several techniques have been proposed to decrease the load-to-use latency. Some of them rely on past history of memory accesses to predict an address and perform a speculative memory request:

Data prefetching cuts the overall load-to-use latency by bringing likely-to-be-used data from distant memory to an upper level of the memory hierarchy ahead of time. Since the goal of data prefetching is only to bring closer the likely-to-be-used data, no recovery schemes are required: a misprediction (i.e. a useless prefetch) does not have any severe impact except for wasted bandwidth and the possible eviction of some data from the memory hierarchy. However, data prefetching does not eliminate the load-to-use latency of the 1st level cache access. Baer and Chen defined the concept of data prefetching and introduced last-address and stride-based prefetchers [3,5].

The goal of address prediction is to speculate on the address of a load instruction during an early stage of the pipeline, in order to initiate the memory access earlier and hide the memory hierarchy latency. [9] proposed to speculate on the address of a load based on a stride predictor. No recovery schemes were required since the data delivery was performed after the address prediction was verified. [14] extended the work by decoupling the verification from the execution of dependent instructions. Dependent instructions are executed based on the speculated loaded data, hence requiring recovery schemes. This study was based on last-address predictors although stride-based address predictors were also mentioned. [11] proposed to share the same stride-based prediction structures to perform both address prediction and data prefetching (next invocations) simultaneously. Finally, [4] introduced the concept of both context-based and correlated load address predictions. They also analyzed the effects of pipelining on the performance of such predictors.

Austin and Sohi [1] proposed caching the SP (stack pointer in MIPS ISA) and GP (global pointer, used to access static variables) registers as well as a number of general-purpose registers used recently as base or index components of memory address. Cached register values are used to speculatively generate a virtual address at the end of the decode stage and access the cache speculatively. A write to the cached register used for address prediction by an instruction currently in the pipeline causes a misprediction. As opposed to zero-cycle loads, register tracking provides an implementable scheme for safe early load address resolution in pipelined out-of-order processor. [2] proposed to speedup address generation by speculatively performing a fast carry-free addition of the index portions of base and offset address components early in the cache access cycle. This scheme saves one cycle of load latency for global-pointer addressing, but is less efficient for stack operations.

Stack cache design was discussed in [8] and recently in [6].

6. Conclusions

In this paper we presented a new mechanism for early memory addresses resolution in the front-end part of the processor pipeline. The proposed scheme allows **safe** computation of memory addresses by utilizing decode-time information on address components. The early load address resolution uses *register value tracking* to compute the values of general-purpose registers by tracking the outcomes of simple operations. The tracking mechanism analyses decoded instruction stream and actually performs the specified arithmetic operation on the internal copy of the architectural register. To simplify the tracking hardware we only keep track of operations of the form *reg±immediate*.

Untrackable register modifications cause the tracking hardware to stop early memory resolution and to wait for the computed register values to be received from the execution core. The register delta value is computed in order to compensate for the register modifications of all in-flight uops.

We have shown that early load address resolution reduces the load-to-use latencies of accessing the data cache and improves the overall performance. Furthermore, since the proposed tracking scheme is safe, successfully tracked uops do not need to be re-executed by the execution core, freeing additional resources for other micro-operations.

Register tracking may be performed in any pipeline stage after instruction decode. Resolving memory references early in the front-end pipeline may be useful to partially hide load-to-use latencies in accessing lower levels of the memory hierarchy. On the down side, longer pipeline gap between the tracking and the execution stages reduces register trackability.

We presented three possible implementations of the tracking scheme that vary in their complexity and potential gain. The simplest mechanism, a*bsolute address tracking* handles about 10% of the loads. It achieves bellow 1% performance gain since it only enables early issue of the loads that do not depend on previous instructions. The second level of tracking is *ESP based tracking* which tackles all stack references. This scheme covers about 24% of the loads and yields the performance gain of 7%. The most aggressive mechanism, the *displacement based tracking*, involves full-blown register tracking. This scheme handles as many as 54% of all loads and can gain over 12% speedup. However this scheme is more complex to implement, and is less efficient in terms of area per gain.

As opposed to address prediction, the proposed scheme is not speculative, and therefore it requires no storage for the previous

accesses history, it does not waste power and cycles due to the processing of speculated data, and it is simple to implement.

Early resolved loads should not be advanced ahead of stores that access the same address. We have shown that a limited, but precise *memory disambiguator* can be constructed using register value tracking. More aggressive schemes may require the usage of other speculative memory disambiguation schemes.

ESP register value tracking enables the implementation of a *stack cache* as an efficient alternative to a multi-ported L1 data cache. A small, multi-ported, low-latency write-back stack cache that works in parallel to a L1 data cache presents a good trade-off. Its lower latency and small cost more than offset the penalties for the rare snoop hits.

We conducted extensive simulations to demonstrate the potential of the proposed mechanisms. We made sure to take into account the implication of the pipelined environment on address tracking.

Future Work. The work on address tracking should continue in several dimensions:

- Combine address prediction and register tracking with unified renaming and memory disambiguation. If done right, this combination can provide very early accesses to data within the memory hierarchy while skipping the memory hierarchy altogether when the data can be retrieved directly from physical registers.
- Extend the concept of trackability beyond address tracking. Register value tracking may be combined with classic value prediction to improve performance by identifying more results and with a better confidence level.

References

[1] T. M. Austin and G. S. Sohi, Zero-cycle Loads: Microarchitecture Support for Reducing Load Latency, in Proceedings of the *28th Annual International Symposium on Microarchitecture*, November 1995.

[2] T.M.Austin, D.N. Pnevmatikatos, G.S. Sohi. Streamlining Data Cache Access with Fast Address Calculation, In *22nd International Symposium on Computer Architecture*, 1995, pp. 369-380

[3] J. Baer and T. Chen, An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty, in Proceedings of the *International Conference on Supercomputing*, November 1991.

[4] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, U. Weiser. Correlated Load Address Predictors, in Proceedings of the *26th Annual International Symposium on Computer Architecture*, May 1999.

[5] T. Chen and and J. Baer, Effective Hardware-Based Data Prefetching for High-Performance Processors, in *IEEE Transactions on Computer, V.45 N.5*, May 1995.

[6] S. Cho, P.-C. Yew, G. Lee. Decoupling Local Variable Accesses in a Wide-Issue Superscalar Processor, in Proceedings of the *26th International Symposium on Computer Architecture*, May 1999.

[7] G. Chrysos and J. Emer, Memory Dependence Prediction Using Store Sets, in Proceedings of the *25th International Symposium on Computer Architecture*, July 1998.

[8] D. Ditzel and R. McLellan. Register Allocation for Free: The C Machine Stack Cache, in *Proc. of the Symposium on Architectural Support for Programming Languages and Operating Systems*, March 1982.

[9] R. J. Eickemeyer and S. Vassiliadis, A Load-Instruction Unit for Pipelined Processors, in *IBM Journal of Research and Development*, July 93.

[10] F. Gabbay and A. Mendelson. The Effect of Instruction Fetch Bandwidth on Value Prediction, in Proceeding of the *25th International Symposium on computer Architecture*, July, 1998.

[11] J. Gonzalez and A. Gonzalez, Speculative Execution via Address Prediction and Data Prefetching, in Proceedings of the *International Conference on Supercomputing*, 1997.

[12] Pentium Pro Family Developer Manual, Volume 2: Programmer s Reference Manual, Intel Corporation, 1996

[13] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, A. Yoaz, A Novel Renaming Scheme to Exploit Value Temporal Locality Through Physical Register Reuse and Unification, in Proceedings of the *31st Annual International Symposium on Microarchitecture*, November 1998.

[14] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, Value Locality and Load Value Prediction, in Proceedings of the *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[15] A. I. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, Speculation and Synchronization of Data Dependencies, in Proceedings of the *24th International Symposium on Computer Architecture*, June 1997.

[16] A. I. Moshovos and G. S. Sohi, Streamlining Inter-operation Memory Communication via Data Dependence Prediction, in Proceedings of the *30th Annual international Symposium on Microarchitecture*, December 1997.

[17] E. Rotenberg, S. Bennett, and J. Smith, Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching, in Proceedings of the *29th International Symposium on Microarchitecture*, December 1996.

[18] R. Valentine, G. Sheaffer, R. Ronen, I. Spillinger and A. Yoaz, Out-of-order Superscalar Microprocessor with a Renaming Device that Maps Instructions from Memory to Registers, *U.S. Patent 5,838,941*, November 1998.

[19] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan, Speculation Techniques for Improving Load Related Instruction Scheduling, in Proceedings of the *26th Annual International Symposium on Computer Architecture*, May 1999.